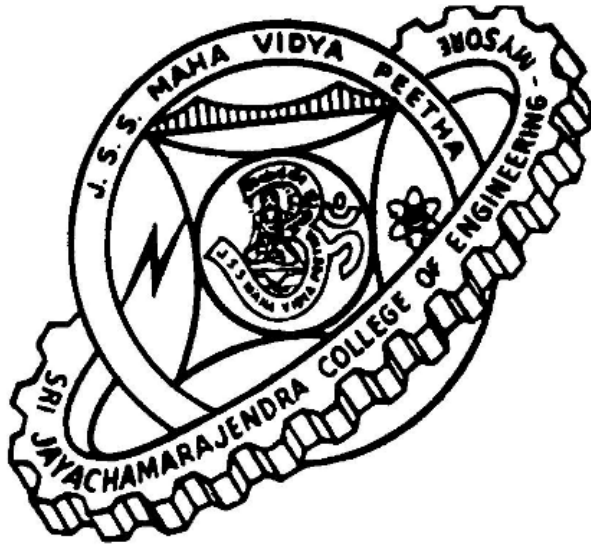

Project Report

On

Implementation of FP-tree



Submitted By,

Pradeep Kashyap R. R.NO 60

Zabee Ulla R.No 65

Submitted to,

Smt. Pushpalatha

Associate Professor

**Department of Computer Science and Engineering,
SJCE, Mysore.**

ABSTRACT

The FP-Growth Algorithm is an alternative algorithm used to find frequent itemsets. It is vastly different from the Apriori Algorithm in that it uses a **FP-tree** to encode the data set and then extract the frequent itemsets from this tree. This section is divided into two main parts, the first deals with the representation of the FP-tree and the second details how frequent itemset generation occurs using this tree and its algorithm. The FP-growth algorithm is currently one of the fastest approaches to frequent item set mining. In this report we describe a C++ implementation of this algorithm, which contains the core operation of computing a projection of an FP-tree (the fundamental data structure of the FP-growth algorithm).

Introduction

Apriori: uses a generate-and-test approach _ generates candidate itemsets and tests if they are frequent

- Generation of candidate itemsets is expensive (in both space and time)
- Support counting is expensive
- Subset checking (computationally expensive)
- Multiple Database scans (I/O)

FP-Growth: allows frequent itemset discovery without candidate itemset generation. Two step approach:

- Step 1: Build a compact data structure called the FP-tree
- Built using 2 passes over the data-set.
- Step 2: Extracts frequent itemsets directly from the FP-tree
- Traversal through FP-Tree.

As we know that, in many cases the apriori candidate generate-and-test method significantly reduces the size of the candidate sets, leading to good performance gain. However, it can suffer from two non-trivial costs

1. It may still need to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
-
-

-
-
2. It may need to repeatedly scan the whole data base and check a large set of candidates by pattern matching. It is costly to go over each transaction in the data base to determine the support of the candidate itemsets.
 3. It number of itemsets generated also depends on the minimum support provided. If the minimum support is small then the join operation performed generates too much of candidate sets, which further adds to the number of generation of the candidate sets. Which may also increase the number of L sets generated. Which is simply waste of computational space and time.

Can we design a method that mines the complete set of frequent itemsets without such a costly candidate generation process is the question which arises. An interesting method in this attempt is called frequent pattern growth, or simply FP-growth, which adopts a divided and conquer strategy as follow. First, it compresses the database representing frequent items into a frequent pattern tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases, each associated with one frequent item or pattern fragment and mines each database separately. For each pattern fragment, only its associated data sets need to be examined. Therefore, this approach may substantially reduce the size of the data sets to be searched, along with the growth of patterns being examined.

The algorithm

The FP-Growth Algorithm is an alternative way to find frequent itemsets without using candidate generations, thus improving performance. For so much it uses a divide-and-conquer

strategy. The core of this method is the usage of a special data structure named frequent-pattern tree (FP-tree), which retains the itemset association information.

In simple words, this algorithm works as follows: first it compresses the input database creating an FP-tree instance to represent frequent items. After this first step it divides the compressed database into a set of conditional databases, each one associated with one frequent pattern. Finally, each such database is mined separately. Using this strategy, the FP-Growth reduces the search costs looking for short patterns recursively and then concatenating them in the long frequent patterns, offering good selectivity.

In large databases, it's not possible to hold the FP-tree in the main memory. A strategy to cope with this problem is to firstly partition the database into a set of smaller databases (called projected databases), and then construct an FP-tree from each of these smaller databases.

The next subsections describe the FP-tree structure and FP-Growth Algorithm, finally an example is presented to make it easier to understand these concepts.

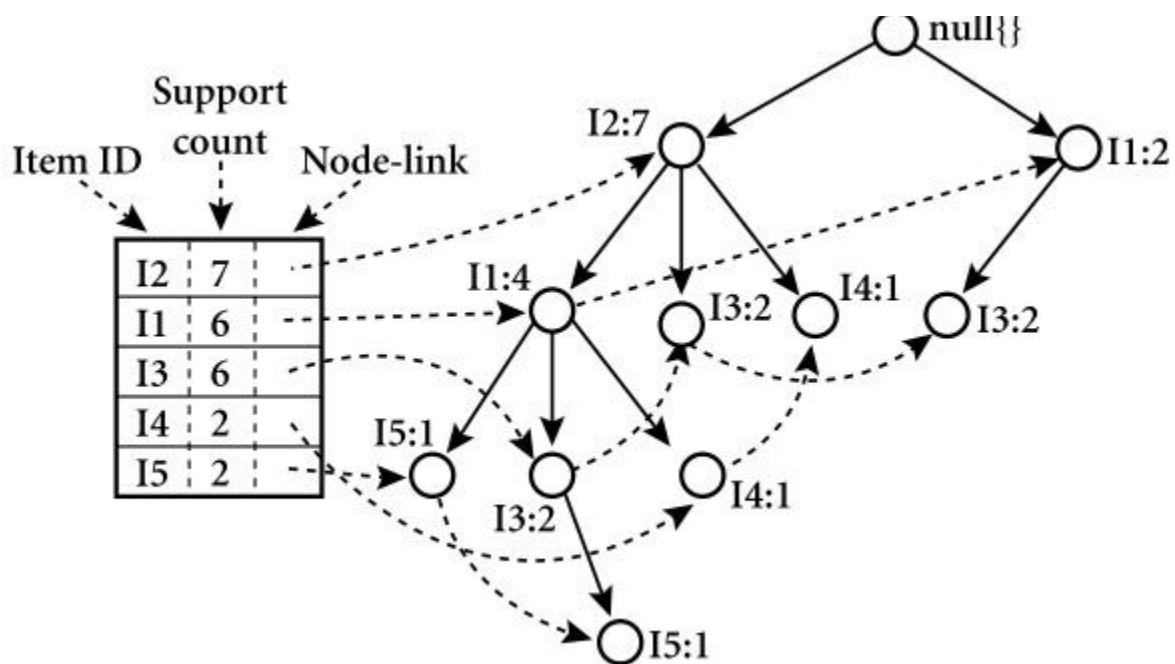
1.1.1 FP-Tree structure

The frequent-pattern tree (FP-tree) is a compact structure that stores quantitative information about frequent patterns in a database.

Han defines the FP-tree as the tree structure defined below:

1. One root labeled as “null” with a set of item-prefix subtrees as children, and a frequent-item-header table (presented in the right side of Figure 1);
 2. Each node in the item-prefix subtree consists of three fields:
 1. Item-name: registers which item is represented by the node;
-
-

2. Count: the number of transactions represented by the portion of the path reaching the node;
 3. Node-link: links to the next node in the FP-tree carrying the same item-name, or null if there is none.
1. Each entry in the frequent-item-header table consists of two fields:
 1. Item-name: as the same to the node;
 2. Head of node-link: a pointer to the first node in the FP-tree carrying the item-name. Additionally the frequent-item-header table can have the count support for an item. The Figure 1 below show an example of a FP-tree.



Algorithm: FP_growth Mine frequent items using an FP-tree by pattern fragement growth

Input:

D, a transaction database;

Min_sup, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps
 - (a) Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the list of frequent items.
 - (b) Create the root of an fp-tree, and label it as null. For each transaction trans in D do the following
 - (c) Select and sort the frequent items in trans according to the order of L. Let the sorted frequent item list in trans be p[P] where p is the first element and P is the remaining list. Call insert_tree(p[P], T), which is performed as follows. If T has a child N such that N.itemname=p.itemname, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its nodelink to the nodes with same itemname via the node-link structure. If P is nonempty, call insert_tree(P,N) recursively.
 2. Procedure FP_growth(Tree,a)
 - If Tree contains a single path then
 - For each combination of the nodes in the path p
-
-

Generate patter b U a with support_count = minimum sup_count

Else for each a_i in the header of tree

Generate patter $b=a_i$ U a with support_count= a_i .support_count;

Construct b's conditional pattern base and then b's conditional FP_tree

tree_b

If (tree_b != NULL) then

Call FP_growth(Tree_b,b);

Class Diagrams

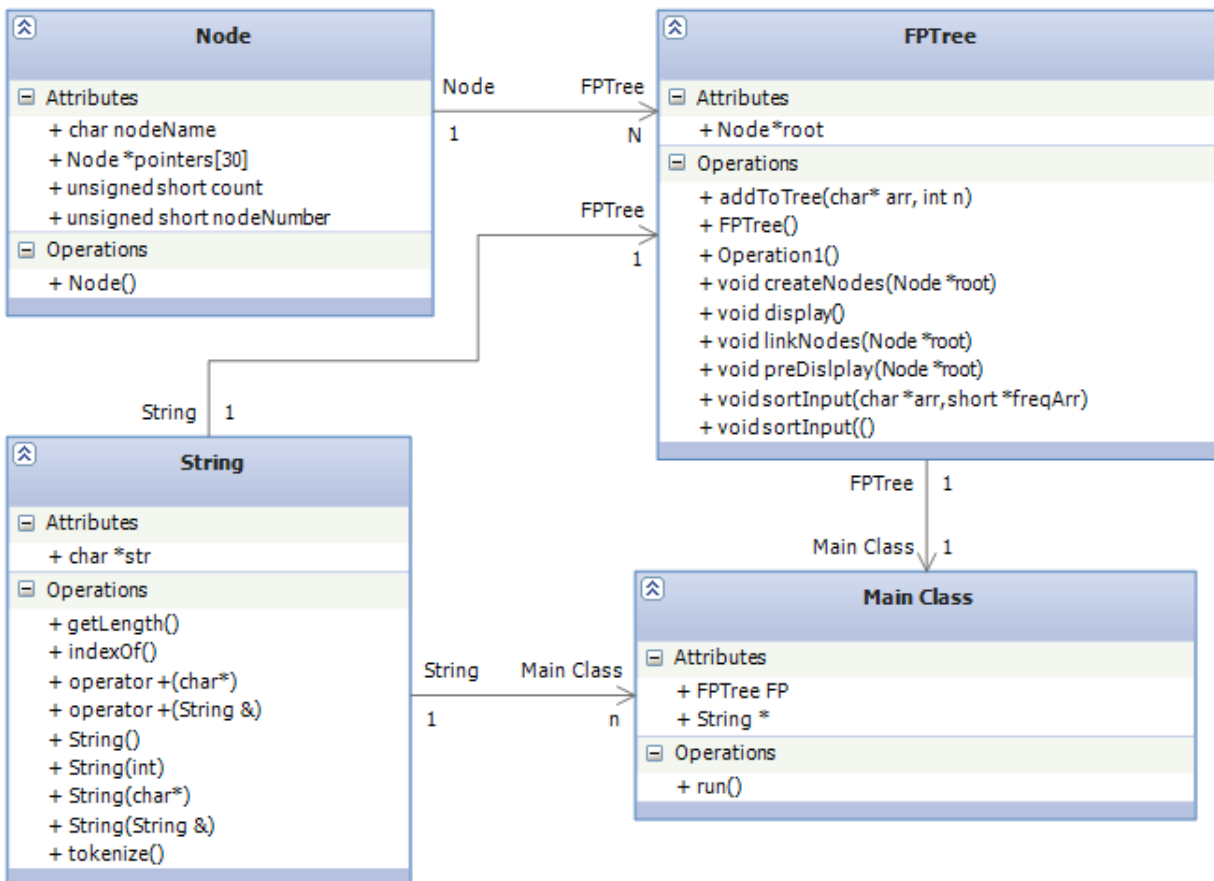


Fig 2. Depicting the class diagrams of the FPTree implementation

The above class diagram depicts the interaction and composition of classes. The node class is the major class which represents the very node in the FPtree. FPtree is a 'n' ary tree. So the number of children for a particular node is not known at compile time. As this implementation considers input aonly from a-z, we require maximum of 26 pointer. The nodeName is the attribute which indicates the character associated with a particular node. As this implementation uses the dotty tool available on Linux platforms, each node in the tree should be represented by a unique number. Hence we have the nodeNumber attribute. The count attribute is used to count the number of instances in which the input is traversed through the node.

The FPtree class constains a pointer called as root which points to the root of the tree. It contains a method which creates nodes according to the input provided. If the node does not exists according to the sorted input with respect to the frequency of each character, then a node is created, else just the count of the node is incremented by one and the input pointer is advanced. The display method just traversed the entire tree recursively and display the value of the each node on to the terminal. The createNode method is used to create all the nodes which are present in the tree to a dotty file. Then the linkNode method will link each of the nodes created. The sortInput method is used to sort the inputs according to the frequency of each character.

String is one of the major class which is extensively used in the implementation. The string class has function for searching a particular pattern in the string, for tokenizing a string and other useful functions. It has special functions which directly convert the integer to string and vice versa. This has overloaded several operators to make it more realistic and interactive.

Implementation Deatails

```
#include<iostream>
#include<stdlib.h>
#include "string.cpp"
#include<fstream>

unsigned short nodeCount=0;
ofstream out;

class node
{
public:
    char nodeName;
    unsigned short count,nodenumber;

    node *pointers[30];
    node()
    {
        nodeName=0;
        count=1;
        nodenumber=0;
        for(int i=0;i<30;i++)
            pointers[i]=NULL;
    }
};

class FPTree
{
public:
    node *root;
    FPTree()
    {
        node *n=new node();
        n->nodeName='0';
        n->nodenumber=0;
        n->count=0;
        nodeCount++;
        root=n;
    }

    void addToTree(char *arr,int n)
    {
        node *temp=root;
        int k=0;
        while(k<n)
        {
            if(temp->pointers[arr[k]-'a']==NULL)
            {
                node *n=new node();
```

```

        n->nodeName=arr[k];
        n->nodenumber=nodeCount;
        nodeCount++;
        temp->pointers[arr[k]-'a']=n;
        temp=n;
    }
    else
    {
        temp=temp->pointers[arr[k]-'a'];
        temp->count++;
    }
    k++;
}

void preDisplay(node *root)
{
    cout<<root->nodeName<<"  "<<root->count<<endl;
    for(int i=0;i<30;i++)
    {
        if(root->pointers[i]!=NULL)
        {
            preDisplay(root->pointers[i]);
        }
    }
}

void createNodes(node *root)
{
    String s,s2,nodecount;
    s=root->nodenumber;
    s2.str[0]=root->nodeName;
    s2.str[1]=0;
    nodecount=root->count;
    s=s + (char*) " [ style=bold label=\"'" + s2 + (char*) "''\n" +
nodecount + (char*) "\" ]\n";
    out.write((char*)s.str,s.getLength());
    out.flush();

    for(int i=0;i<30;i++)
    {
        if(root->pointers[i]!=NULL)
        {
            createNodes(root->pointers[i]);
        }
    }
}

void linkNodes(node *root)
{
    String s,s2,temp;
    s=root->nodenumber;

    for(int i=0;i<30;i++)
    {
        if(root->pointers[i]!=NULL)
        {
            s2=root->pointers[i]->nodenumber;

```

```

        temp=s + (char*)"->" + s2 + (char*)"\n";
        out.write((char*)temp.str,temp.getLength());
        out.flush();
        linkNodes(root->pointers[i]);
    }
}

void display()
{
    node * temp=root;
    preDisplay(temp);
    createNodes(temp);
    linkNodes(temp);
}

};

void sortInput(char *arr, unsigned short freqArr[])
{
    int k=strlen(arr);
    for(int i=0;i<k-1;i++)
        for(int j=0;j<k-1;j++)
            if(freqArr[arr[j]]<freqArr[arr[j+1]])
            {
                char temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }

    for(int i=0;i<k;i++)
        cout<< arr[i]<<" ";
    cout<<endl;
}

int main()
{
    String s=String((char*)"digraph dg \n{\n");
    out.open("fptree.dot",ios::binary);
    out.write((char*)s.str,strlen(s.str));
    out.flush();
    FPTree fpt=FPTree();
    char arr[30][30]={0};
    int n, no_items=0;
    unsigned short freqArr[256]={0};
    cin>>n;

    for(int i=0;i<n;i++)
    {
        int j=0;
        cin>> no_items;
        for(j=0;j<no_items;j++)
        {
            cin>>arr[i][j];
            freqArr[arr[i][j]]++;
        }
        arr[i][j]='\0';
    }
}

```

```

        for(int i=0;i<256;i++)
        {
            if(freqArr[i]!=0)
                cout<<char(i)<<"    : "<<freqArr[i]<<endl;
        }

    for(int i=0;i<n;i++)
        sortInput(arr[i],freqArr);

    for(int i=0;i<n;i++)
        fpt.addToTree(arr[i],strlen(arr[i]));
    fpt.display();
    s=(char*)"\\n";
    out.write((char*)s.str,strlen(s.str));
    out.close();
    system("dotty fptree.dot");
    return 0;
}

/*This program includes
    1. Operator overloading for reading and writing an object from console
    2. Multiple types of constructors
    3. Dynamic memory allocation
    4. Illustration of THIS pointer
    5. Nesting of member functions
    6. Anonymous object creation. (NOT possible in turbo c++)
    7. Writing of String object to file as text.
    8. Error handling. Exception handling (NOT possible in turbo c++)
    9. Function overloading
    10. Conversion from class to basic
    11. Conversion from basic to class
    12. single Operator cascading.
    13. Multiple Operator cascading.
    14. Extensive string handling functions defined.
*/

#include<iostream>
#include<fstream>
#include<cstring>
#include<cstdlib>
#include<exception>

using namespace std;

class String
{
    char delim;                //Deliminter for reading the
    string from the console
    int length;
    int find(String,char);     //Stores the length of
    the string

    public:
        char *str;             //Character array

```

```

        short instructionAddress; //Only used as a special data member for
sic assembler
        //Default constructordd
        //default delimiter is newline character. i.e Reads the string till the
new
        String()
        {
            instructionAddress=length=0;
            str=new char[length+1];
            str[0]='\0';
            delim='\n';

        }

        //constructor which constructs(memory allocation) the string with the
specified no of character size.
        String(int size)
        {
            length=size;
            str=new char[length+1];
            str[0]='\0';
            delim='\n';
        }

        //Constructor what takes String literals to construct the object.
        String(char *s)
        {
            length=strlen(s);
            str=new char[length+1];
            strcpy(str,s);
            delim='\n';
        }

        String(const String &s)
        {
            str=new char[strlen(s.str)+1];
            strcpy(str,s.str);
            delim='\n';
        }

        //this funtion sets the delimeter for reading the string from the key
borad
        void setDelimiter(char ch)
        {
            delim=ch;
        }

        //Retruns the length of the String object. (no of character exculding
'\0' character
        int getLength()
        {
            this->length=strlen(str);
            return length;
        }

        //WRITING STRING TO CONSOLE "<<" overloaded
        friend ostream & operator <<(ostream &out,String s)

```

```

{
    out<<s.str;
    return out;
}

//WRITING STRING TO FILE
friend ostream& operator <<(ostream &out, String s)
{
    out<<s.str;
    return out;
}

//READING STRING FROM FILE
friend ifstream & operator >>(ifstream &fin,String &s)
{
    s=String(1024);
    fin>>s.str;
    return fin;
}

//Converts the string object to character array (Returns the address of
the pointer which points to the
//                                beggining of the
character array)
char * toCharArray()
{
    return str;
}

//SUBSTRING 1 TYPE (which gives string from the specified location till
the end of string)
String subString(int index)
{
    if(index>strlen(str) || index<0)
    {
        throw ("Array out of bound exception!"); // if index is out of
range(array bound cheking).
    }
    return *new String(&str[index]);
}

//SUBSTRING 2 TYPE (which gives the substring from the specified
starting location
//                                to the location specified by the "end" variable
String subString(int start,int end)
{
    if( start<0 || end > strlen(str) || start>end)
    {
        throw("Array out of bound exception!");//Array bound cheking.
    }

    String s(this->getLength());
    strcpy(s.str,str);
    s.str[end+1]='\0';
    s.str=&s.str[start];
}

```

```

        return s;
    }

    // Function prototypes for which definition is available outside the
class.
    friend istream & operator >>(istream &in, String &s);
    operator int();

    String operator + (char *);
    String operator + (String);
    String operator - (char *);
    String operator - (String);
    int operator ==(String);
    int operator ==(char *);
    int toInteger();
    void operator =(int);
    String * tokenize(char,unsigned int&);
    int indexOf(String );
    String trim(char ch=' ');
};

//Function for searching a string in the text..
int String::indexOf(String pattern)
{
    int j=0;
    if(pattern.getLength() > this->getLength())
    {
        return -1;        // Error that the text to be searched cannot be
more than the text.
    }
    for(int i=0;i<=this->getLength()-pattern.getLength();i++)
    {
        j=0; //To start searching pattern in the text from beggining of
pattern.(text index is continued).
        while(str[i]==pattern.str[j] && j<pattern.getLength())//till
characters are successively found and till j< patterns's length
        {
            i++;
            j++;
        }
        if(j==pattern.getLength())
        {
            return i-j; //return the location where the character found.
        }
    }
    return -1; //NO match found
}

// This function is used to remove the unwanted character and tabspace
// before and after the string. (it also removes the tab spaces in
between the text, but not the spaces *)...
String String:: trim(char ch)
{
    String temp=*this;
    for(int i=0;i<temp.getLength();i++)
    {

```

```

        if(temp.str[i]=='\t')
        {
            temp.str[i]=ch;           //replacing tabspace by single space.
        }
    }
    int beg=0;
    char chr=temp.str[beg];
    while(chr==ch)    // This code(while part) is used to remove the blank
spaces at the beggining.
    {
        beg++;
        chr=temp.str[beg];
    }
    temp=temp.subString(beg);

    int end=temp.getLength()-1;
    chr=temp.str[temp.getLength()-1];
    while(chr==ch) // Used to remove the blank spaces at the end.
    {
        end--;
        chr=temp.str[end];
    }
    temp=temp.subString(0,end);
    *this=temp;
    return *this;
}

```

```

// One of the most widely used string fucntions
/* This function is used to tokenize(create fragments or parts) the given
string
    into array of string with the given deliminter. i.e "ch" here.
(excluding delimiter).
*/

```

```

//READING STRING FROM CONSOLE
istream & operator >>(istream &in,String &s)
{
    char ch,temp=s.delim;
    s=String(1024);
    s.delim=temp;
    int i=0;
    while((ch=getchar())!=s.delim)//read the character till the
delimiter character.
    {
        s.str[i]=ch;
        i++;
    }
    s.str[i]='\0';
    s.length=strlen(s.str);
    return in;
}

```

```

void stringReverse(char *s)
{
    char temp[strlen(s)+1];
    int i=0;
    for(i=0;i<strlen(s);i++)

```

```

        {
            temp[i]=s[strlen(s)-1-i];
        }
        temp[i]='\0';

        for(i=0;i<strlen(s);i++)
        {
            s[i]=temp[i];
        }
    }

    //Conversion from basic to class type
    void String::operator=(int i)
    {
        String temp=String(20);
        if(i==0)
        {
            temp.str[0]='\0';
            temp.str[1]=0; // null char at the end
            *this=temp;
            return;
        }

        int rev=0,mul=10;
        temp.str[0]='\0';
        int k=0;
        while(i>0)
        {
            if(!((i%10)+48)>=48 && ((i%10)+48)<=57) //checking whether the
            character is not b/w 30h and 39h, if so throw the error.
            {
                //cout<<"Conversion exception\n Invalid character occurrence while
            conversion\n"<< i<<endl;
                throw("Conversion exception\n Invalid character occurrence while
            conversion\n");
            }
            temp.str[k]= (i%10)+48;
            i/=10;
            k++;
        }
        /* i=0;
        while(rev!=0)
        {
            temp.str[i]= char((rev%10) + 48); //converts integer to string.
            rev/=10;
            i++;
        }

        temp.str[i]='\0';
        */
        temp.str[k]='\0';
        stringReverse(temp.str);
        *this=temp;
    }
    //converts from string to integer.
    int String :: toInteger()

```

```

    {
        char *temp=str;
        int res=0,mul=1;
        while(*temp!='\0')
        {
            temp++;
        }
        temp--;

        while(temp!=(str-1))
        {
            if(int(*temp==7))
            {
                temp--;
                continue;
            }
            if( int(*temp) <48 || int(*temp) >57) //checking for non numerical
character if found throw an exception.
            {
                throw ("Invalid string to integer conversion\n Invalid characters
found\n");
            }
            else
            {
                res=res+ mul*(int(*temp)-int('0'));
                mul*=10;
            }
            temp--;
        }

        return res;
    }

String :: operator int()
{
    return this->toInteger();
}

//For string concatenation with string literal
String String:: operator + (char *s)
{
    String temp=String();
    temp.str=new char[ strlen(this->str) + strlen(s) +1];
    strcpy(temp.str,this->str);
    strcat(temp.str,s);

    return temp;
}

//For String concatenation with another string object.
String String:: operator +(String s)
{
    String temp=String();
    temp.str=new char[ strlen(this->str) + strlen(s.str) +1];
    strcpy(temp.str,this->str);
    strcat(temp.str,s.str);
}

```

```

        return temp;
    }

//For deleting the character that are there in *s from the string object
String String::operator -(char *s)
{
    String temp(this->getLength());
    int j=0;

    String del(s);

    for(int i=0;i<this->getLength();i++)
    {
        if(find(del,str[i]))
        {
            continue;
        }
        else
        {
            temp.str[j]=str[i];
            j++;
        }
    }
    temp.str[j]='\0';
    temp.length=strlen(temp.str);
    return temp;
}

//Finds a character in a given string. (used for only internal operational
purpose and is hidden to user.
//As they have indexOf function for searching.
int String:: find(String s,char ch)
{
    for(int i=0;i<s.getLength();i++)
    {
        if(s.str[i]==ch)
            return 1;
    }
    return 0;
}

//For deleting the character that are there in string object s from the
string object
String String::operator -(String s)
{
    return *this- s.str; // NEsting of member functions
}

int String:: operator == (String s)
{
    if(this->getLength() != s.getLength())
    {
        return 0;
    }
}

```

```

    for(int i=0;i<s.getLength();i++)
    {
        if(this->str[i] != s.str[i])
        {
            return 0;
        }
    }
    return 1;
}

int String:: operator ==(char *s)
{
    if(this->getLength() != strlen(s))
    {
        return 0;
    }

    for(int i=0;i<strlen(s);i++)
    {
        if(this->str[i] != s[i])
        {
            return 0;
        }
    }
    return 1;
}

int countTokens(char *str,char delimiter)
{
    char ch=0;
    int no_tokens=0;
    int k=0;
    ch=str[k];
    while(ch!='\0')
    {
        while(ch==delimiter)
            ch=str[++k];

        if(ch=='\0')
            break;

        no_tokens++;

        while(ch!=delimiter && ch!='\0')
            ch=str[++k];

        if(ch=='\0')
            break;
    }
    return no_tokens;
}

String * String::tokenize(char delimiter,unsigned int &n)
{
    n=countTokens(str,delimiter);
    if ( n==0 )

```

```

        return NULL;

String *tokens=new String[n+1];
unsigned int start=0,i=0,end=0,k=0, tokenIndex=0;
char ch=0;
ch=str[k];

while(ch!='\0')
{
    while(ch==delimiter)
        ch=str[++k];

    if(ch=='\0')
        break;

    start=k;

    while(ch!=delimiter && ch!='\0')
        ch=str[++k];
    end=k;

    tokens[tokenIndex].str=new char[end-start+2];
    for(i=0;i<end-start;i++)
        tokens[tokenIndex].str[i]=str[i+start];
    tokens[tokenIndex].str[i]='\0';
    tokenIndex++;
}
return tokens;
}

/* String * String:: tokenize(char ch,int &n)
{
    if(getLength()==0)//checking for empty string
    {
        n=0;
        return 0;
    }
    String temp(this->getLength());

    strcpy(temp.str,this->str);
    temp.trim(ch);
    int count=0,j=0,i,k;
    int indexes[temp.getLength()];

    for(i=0;i<temp.getLength();i++)
    {
        if(temp.str[i]==ch)
        {
            count++;
            indexes[j]=i;           //count the number of occurrence of the
delimiter.
            j++;
            while(temp.str[i]==ch)
                i++;
        }
    }
    indexes[j]=temp.getLength()-1;//assign even the last character's index.
}

```

```
String *token=new String[count]; //create array of string for the number
of tokens.
```

```
    for(k=0;k<=j;k++)
    {
        try
        {
            if(k==0)
            {
                token[k]=temp.subString(0,indexes[k]-1); // assign the
string(as token) from begging to the first occurance // of the
delimiter.
            }
            else
            if(k==j)
            {
                token[k]=temp.subString(indexes[k-1]+1,indexes[k]); //assign the
string from last occurance of the delimiter till the end
            }
            else
            {
                token[k]=temp.subString(indexes[k-1]+1,indexes[k]-1); //assign the
string b/w two delimiters.
            }
        }
        catch(const char *s)
        {}
    }
```

```
    //There will be many occurance of successive delimiters. So for them
empty tokens will be
```

```
    // create. SO they should be removed.
```

```
    int fin_count=0;
```

```
    for(k=0;k<=j;k++)
    {
        if(token[k].getLength()!=0)
        {
            fin_count++; //count only the tokens which have length more than
0.
        }
    }
```

```
    String *fin_token=new String[fin_count]; //create array of string only
for the valid tokens.
```

```
    i=0;
```

```
    for(k=0;k<=j;k++)
```

```
    {
        if(token[k].getLength()!=0)
        {
            token[k].trim(ch);
            fin_token[i]=token[k];
            i++;
        }
    }
```

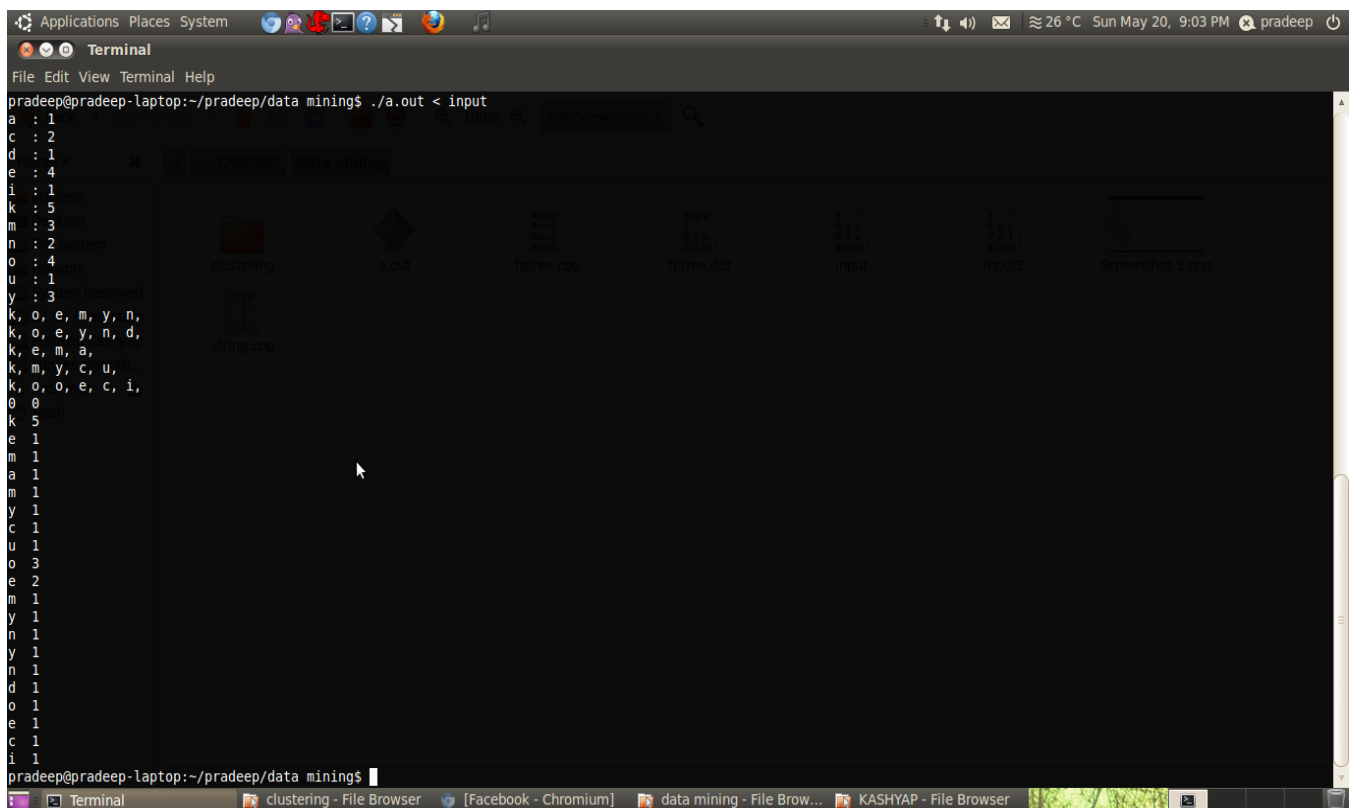
```
    n=fin_count;
```

```
    return fin_token;
```

```
}  
*/
```

The entire code for the implementation is written in the C++ on Linux platform. The compiler used is g++. This program takes input from a file or from the terminal. The file should contain the number of transactions in the first line and next 'n' lines follow. The first input in each line should be the number of items in the transaction followed by the items respectively. The output from the program is a file which is the doty specification. This contains the definition of each node and the link between each of them. The tree is initially constructed and then it is traversed to create nodes for the doty specification. After that again the tree is traversed to get the links between the parent and the children nodes.

Screen Shots



```
pradeep@pradeep-laptop:~/pradeep/data minings$ ./a.out < input  
a : 1  
c : 2  
d : 1  
e : 4  
i : 1  
k : 5  
m : 3  
n : 2  
o : 4  
u : 1  
y : 3  
k, o, e, m, y, n,  
k, o, e, y, n, d,  
k, e, m, a,  
k, m, y, c, u,  
k, o, o, e, c, i,  
0 0  
k 5  
e 1  
m 1  
a 1  
m 1  
y 1  
c 1  
u 1  
o 3  
e 2  
m 1  
y 1  
n 1  
y 1  
n 1  
d 1  
o 1  
e 1  
c 1  
i 1  
pradeep@pradeep-laptop:~/pradeep/data minings$
```

Fig 2. The display of the fptree on the terminal

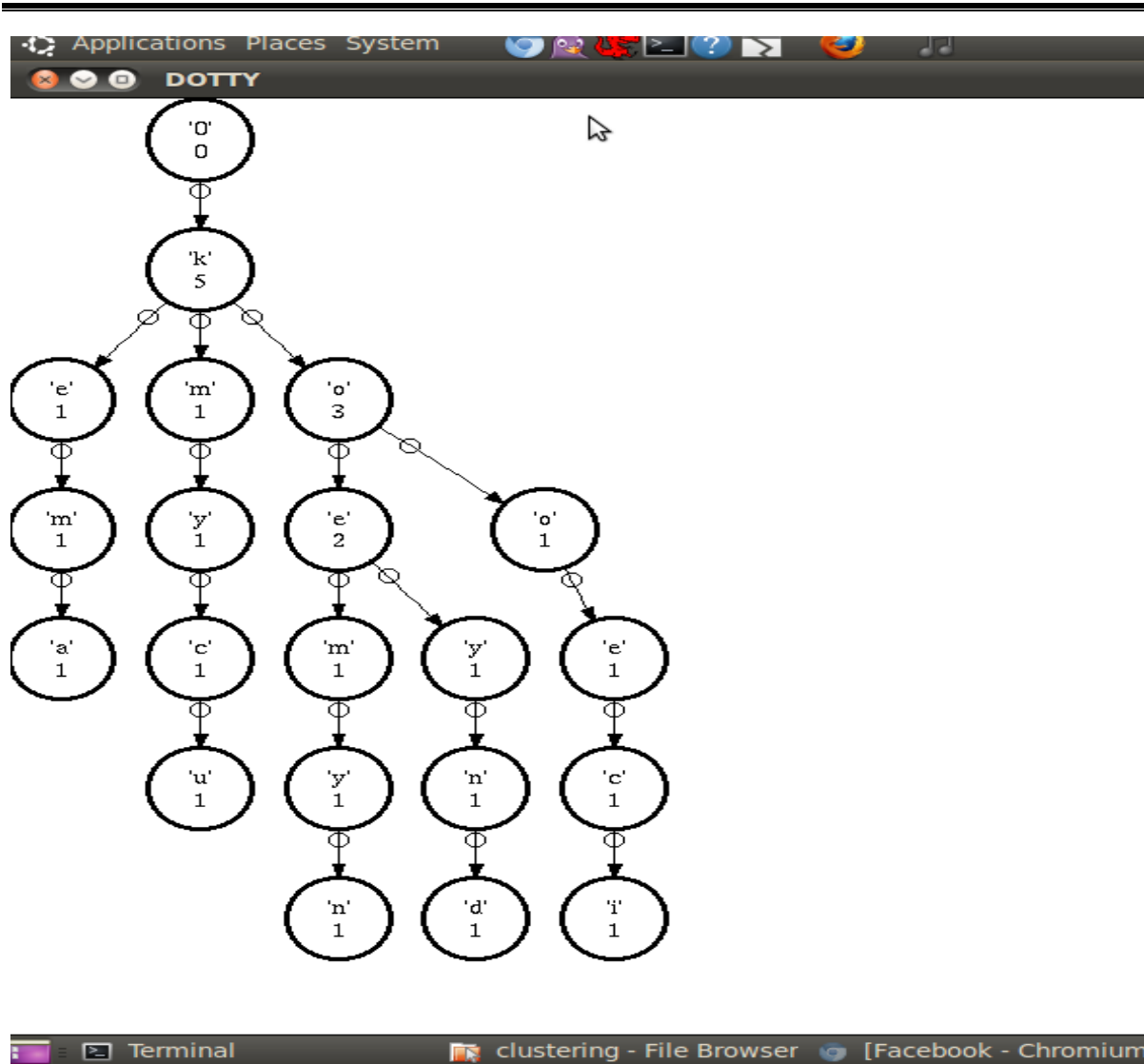


fig 4. Output from dotty program for input file "input"

File "INPUT"

5

6 m o n k e y

6 d o n k e y

4 m a k e

5 m u c k y

6 c o o k i e

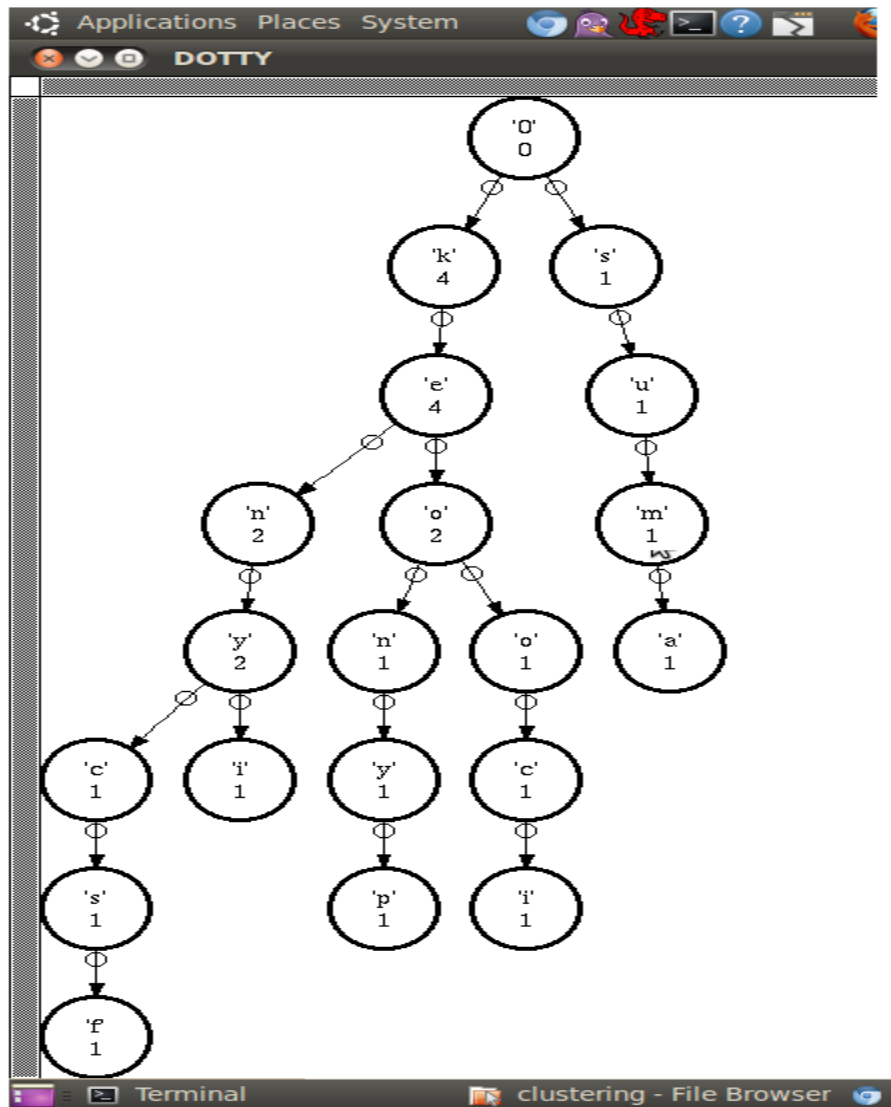


fig 5. Output from dotty program for input file "input2"

FILE "INPUT2"

5

6ponkey

7cfnkeys

4suma

5nikey

6cookie

Conclusion

The FP-Growth Algorithm is an alternative algorithm used to find frequent itemsets. It is used for overcoming the disadvantage of Apriori algorithm. Data set are encoded and then frequent itemsets from the tree is extracted. The implementation of two main parts have been done, the first deals with the representation of the FP-tree and the second details how frequent itemset generation occurs using this tree and its algorithm. In this report we described a C++ implementation of this algorithm, which contains the core operation of computing a projection of an FP-tree.

References

http://en.wikipedia.org/wiki/Association_rule_learning

http://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Frequent_Pattern_Mining/The_FP-Growth_Algorithm

http://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&ved=0CGYQFjAD&url=http%3A%2F%2Fselab.iecs.fcu.edu.tw%2Fwiki%2Fimages%2Fe%2Fe2%2FHigh-utility_item_sets%253B_Pattern_mining%253B_Partition_tree.ppt&ei=KCC5T5LKFMrKrAeQmeDiBw&usg=AFQjCNGq8nqniZUXqBIZ0JhgFVEoz94drQ&sig2=gBYtpRGLHDvHF6y2QJ1T8Q
